

Deep Dive into React Stream/Serialize

2024/04/30 Node学園

Hajime Mugishima / Cybozu Inc.

Hajime Mugishima

**Cybozu Inc.
Frontend Expert Team**

X @mugi_uno



前提知識

React と Stream

- React v18 以降、Stream 経由でのレンダリングをサポート
 - Node Stream : `renderToPipeableStream`
 - Web Stream : `renderToReadableStream`
- App Router でも内部的に利用している

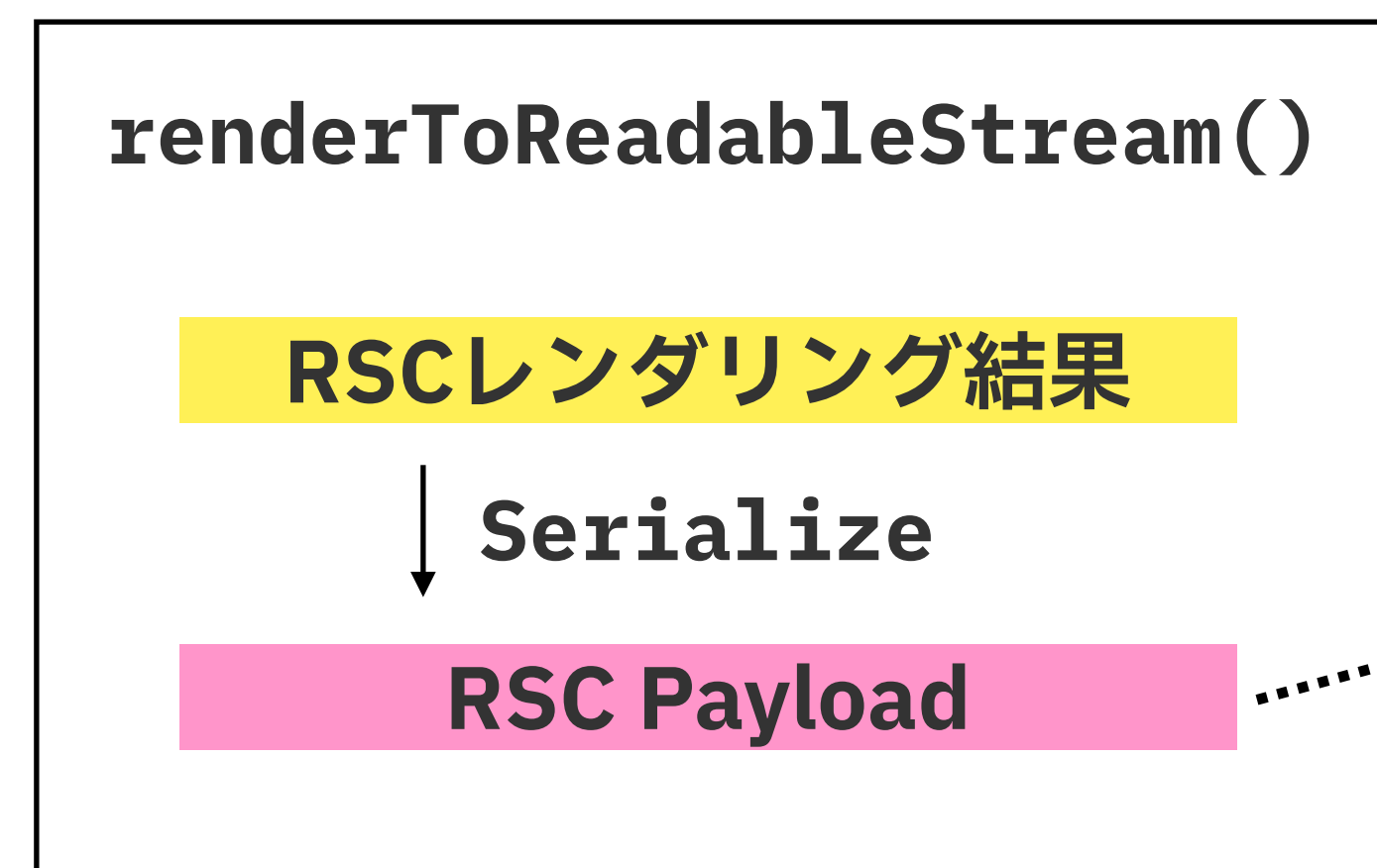
```
const flightReadableStream = renderToReadableStream(  
  options  
    ? [options.actionResult, buildIdFlightDataPair]  
    : buildIdFlightDataPair,  
  ctx.clientReferenceManifest.clientModules,  
  {  
    onError: ctx.flightDataRendererErrorHandler,  
  }  
)
```

vercel/next.js - src/server/app-render/app-render.tsx

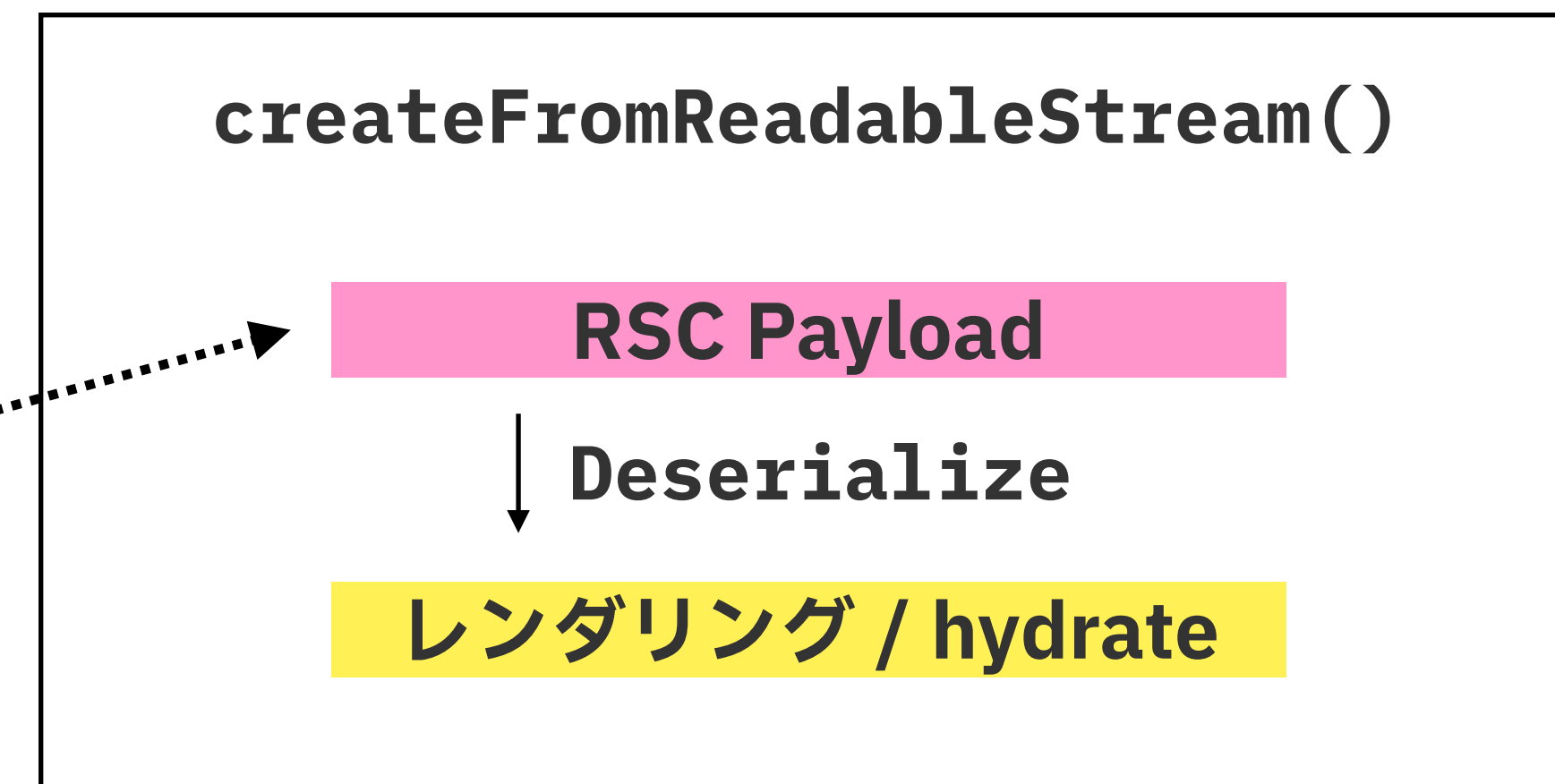
RSC Payload と Serialize/Deserialize

- Server Component のレンダリング結果は RSC Payload と呼ばれる形で Stream に渡される
- 一部のデータは独自の形式で Serialize/Deserialize される (e.g. Client Component に渡される Props など)

Server



Browser



簡単なサンプルアプリを用意してみる

- Async Component のページ
- 2つの Client Component
 - Promiseを渡す
 - 片方は成功、片方は失敗
 - use() で Promise を待機
- <RefreshButton>
→ router.refresh() を呼ぶボタン

```
export default async function Page() {
  await new Promise((resolve) => setTimeout(resolve, 3000));

  const promise = new Promise((resolve) =>
    setTimeout(() => resolve("PROMISE RESOLVED"), 1000)
  );

  const errorPromise = new Promise((_, reject) =>
    setTimeout(() => reject("PROMISE REJECTED"), 2000)
  );

  return (
    <main>
      <Suspense>
        <SuccessClient
          description={" ".padStart(2000, "abcde")}
          date={new Date()}
          promise={promise}
        />
      </Suspense>
      <ErrorClient promise={errorPromise} />
      <RefreshButton />
    </main>
  );
}
```

Stream の中身は？

- `router.refresh()` の後に DevTools で確認

[illegible]

Stream の中身は？

- `router.refresh()` の後に DevTools で確認

	X	Headers	Payload	Preview	Response	Initiator	Timing	Cookies
1					3:I[5613,[],""]			
2					4:I[1778,[],""]			
3					0:["Hd4i-EanmGD0JpIffHyHC",[["","",{"children":["sample",{"children":["__PAGE__",{}]}]},{"\$undefined","\$undef			
4					5:[["\$","meta","0",{"name":"viewport","content":"width=device-width,·initial-scale=1"}],["\$","meta","1",{"			
5					1:null			
6					6:"\$Sreact.suspense"			
7					7:I[9842,["360","static/chunks/app/sample/page-9a95576bd3d960ac.js"],"SuccessClient"]			
8					a:I[5402,["360","static/chunks/app/sample/page-9a95576bd3d960ac.js"],">ErrorClient"]			
9					c:I[5094,["360","static/chunks/app/sample/page-9a95576bd3d960ac.js"],"RefreshButton"]			
10					8:T7d0,abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcd			
11					9:"PROMISE·RESOLVED"			
12					b:E{"digest":"193452068"} 			
13								

呪文…？

このセッションのテーマ

- Stream の中身がなんとなく読めるようになる
- Serialize/Deserialize が何をやってるかちょっとわかる
- 私も最近までは呪文に見えてたのできっと大丈夫

注意点

- App Router 回ですが、実はほぼ React の話です
- おそらく実務ですぐに役立つ話ではありません
(デバッグとかでたまに使えるかも)

Stream データの基礎知識

Stream データの基礎知識

```

3:I[5613,[],"" ]
4:I[1778,[],"" ]
0:["Hd4i-EanmGD0JpIffHyHC",[["",{ "children":["sample",{ "children":["__PAGE__",{ }}]}]},{ "$unde
5:[["$","meta","0",{ "name":"viewport","content":"width=device-width, initial-scale=1"}],["$
1:null
6:"$Sreact.suspense"
7:I[9842,["360","static/chunks/app/sample/page-9a95576bd3d960ac.js"],"SuccessClient"]
a:I[5402,["360","static/chunks/app/sample/page-9a95576bd3d960ac.js"],"ErrorClient"]
c:I[5094,["360","static/chunks/app/sample/page-9a95576bd3d960ac.js"],"RefreshButton"]
8:T7d0,abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcde
9:"PROMISE RESOLVED"
b:E{"digest":"193452068"}

```

Stream データの基礎知識

Chunk

3:I[5613,[],""]

4:I[1778,[],""]

0:["Hd4i-EanmGD0JpIffHyHC",[["",{"children":["sample",{"children":["__PAGE__",{}]}]}],"\$unde

5:[["\$","meta","0",{"name":"viewport","content":"width=device-width, initial-scale=1"}],["\$

1:null

6:"\$Sreact.suspense"

7:I[9842,["360","static/c

a:I[5402,["360","static/c

c:I[5094,["360","static/c

8:T7d0,abcdeabcdeabcdeabc

9:"PROMISE-RESOLVED"

b:E{"digest":"193452068"}

- 基本は 1行に1Chunk
- ID + ":" + Tag + Row の構成

Stream データの基礎知識

- 1行に複数Chunkが含まれるケースもある
- バイナリデータの場合はサイズが指定されており、後続チャンクが連結している
- DevToolsなどで見る場合には注意が必要

- 1行に複数Chunkが含まれるケースもある
- バイナリデータの場合はサイズが指定されており、後続チャンクが連結している
- DevToolsなどで見える場合には注意が必要

```
deabcdeabcdeabcde2:["$","main",null,{"children":["$","$6",null,{"children":["$",""
```

ID=2のChunkがID=8のChunkの末尾と連結

Stream データの基礎知識

ID

```
3:
4:
0: 0]pIffHyHC",[[["
5: ["$", "meta", "0", {"name": "vie
1: null
6: "$react.suspense"
7: [9842, ["360", "static/chunks/
a: [5402, ["360", "static/chunks/
c: [5094, ["360", "static/chunks/
8: 7d0, abcdeabcdeabcdeabcdeabcde
9: PROMISE-RESOLVED"
b: {"digest": "193452068"}
```

- Chunk を一意に特定するキー
- 16進数値
- 参照時の紐づけに利用
- 順序は不定

Stream データの基礎知識

Tag

```
3 I|5613, [], ""]  
4 I|1778, [], ""]  
0: I|Hd4i-EanmGD0  
meta", "  
6: ssreact.suspe  
7 I|9842, ["360",  
a I|5402, ["360",  
c I|5094, ["360",  
8 T|d0, abcdeabcde  
9: "PROMISE- RESOLV  
b E- "digest": "193
```

- データの種別/取り扱い方を表す
- I, H, T, E, D, W, R, P など多くが存在
- 多くは頭文字 (I = Import, D = Debug など)
- 無い場合は Reactツリー関連か単純なデータ (コード上では “Model” と呼ばれる)

Stream データの基礎知識

- データ本体
- Tag に応じて処理される
- 大半は JSON String

[illegible]

Row (or Chunk)

Zero Chunk (便宜上、本資料ではこう呼びます)

- ID=0 の Model Chunk がレンダリングの起点
- Client側での描画 ÷ Zero Chunk の描画
- 読むときも ID=0 の Chunk から見ていくと良い

```
const initialServerResponse = createFromReadableStream(readable, {
  callServer,
})
```

```
function ServerRoot(): React.ReactNode {
  return use(initialServerResponse)
}
```

```
function createFromReadableStream<T>(
  stream: ReadableStream,
  options?: Options,
): Thenable<T> {
  const response: FlightResponse = createResponseFromOptions(options);
  startReadingFromStream(response, stream);
  return getRoot(response);
}
```

```
export function getRoot<T>(response: Response): Thenable<T> {
  const chunk = getChunk(response, 0);
  return (chunk: any);
}
```

Stream Tag の例

"I" → Import

- クライアントコンポーネントの Import 用 Chunk
- webpack であれば `__webpack_require__` が呼ばれる
- Import すべきファイルやコンポーネント情報を含む

```
7: I[9842, ["360", "static/chunks/app/sample/  
page-7f3480f94493b7cb.js"], "SuccessClient"]
```

```
[  
  9842,  
  ["360", "static/chunks/app/sample/page-7f3480f94493b7cb.js"],  
  "SuccessClient"  
]
```

※rowを整形したもの

"E" → Error

- エラー情報 Chunk
- Production 時は digest のみとなる

Production

```
b:E{"digest":"193452068"}
```

Development

```
b:E{"digest":"193452068","message":"Error: PROMISE  
REJECTED","stack":""}
```

"T" → Text

- 1024byteより大きい文字列は別のChunkとなる
- "T"の直後にはサイズが16進数値で付与される
- ↓ は `"".padStart(2000, "abcde")` を Props に渡した場合の例

8: T7d0 , abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeab
cdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdea
bcdeabcdeabcdeabcdeabcdeabcdeabcde . . . (省略)

※"7d0" → 2000

"T" → Text

- Chunk を分ける理由はコード上にコメントに書いてある

```
5     ... }
6     ... if (value.length >= 1024) {
7     ...     // For large strings, we encode them outside the JSON payload so that we
8     ...     // don't have to double encode and double parse the strings. This can also
9     ...     // be more compact in case the string has a lot of escaped characters.
10    ...     return serializeLargeTextString(request, value);
11    ... }
12    ... return escapeStringValue(value);
13    ... }
```

- JSONとの二重エンコード/パースを避けるため
- エスケープが多い文字列の場合にコンパクトにするため

StreamとSerialize

Serialize / Deserialize の基本

- Model Chunk の Serialize / Deserialize は、
実は JSON.stringify および JSON.parse を呼んでいるだけ
- それぞれ replacer / reviver を渡して変換処理をカスタマイズしている

packages/react-server/src/ReactFlightServer.js ※Serialize側

```
const stringify = JSON.stringify;
```

```
// For anything else we need to try to serialize it using JSON.  
// $FlowFixMe[incompatible-type] stringify can return null for undefined but we never do  
const json: string = stringify(value, task.toJSON);  
emitModelChunk(request, task.id, json);  
}
```

packages/react-client/src/ReactFlightClient.js ※Deserialize側

```
function parseModel<T>(response: Response, json: UninitializedModel): T {  
  return JSON.parse(json, response._fromJSON);  
}
```

Zero Chunk を覗いてみる

```
[
  "Hd4i-EanmGD0JpIffHyHC",
  [
    [
      [
        "",
        { "children": ["sample", { "children": ["__PAGE__", {}] }] },
        "$undefined",
        "$undefined",
        true
      ],
      [
        "",
        {
          "children": [
            "sample",
            { "children": ["__PAGE__", {}, ["$L1", "$L2", null]] },
            [
              "$",
              "$L3",
              null,
              {
                ...
              }
            ]
          ]
        }
      ]
    ]
  ]
]
```

※可読性のため整形したもの

Zero Chunk を覗いてみる

```
[
  "Hd4i-EanmGD0JpIffHyHC",
  [
    [
      [
        "",
        { "children": ["sample", { "children": ["__PAGE__", {}] }] },
        "$undefined",
        "$undefined",
        true
      ],
      [
        "",
        {
          "children": [
            "sample",
            { "children": ["__PAGE__", {}, ["$L1", "$L2", null]] },
            [
              "$",
              "$L3",
              null,
              {
                ...
              }
            ]
          ]
        }
      ]
    ]
  ]
]
```

- 全体は JSON String

Zero Chunk を覗いてみる

```
[  
  "Hd4i-EanmGD0JpIffHyHC",  
  [  
    [  
      [  
        "",  
        { "children": ["sa  
        "$undefined",  
        "$undefined",  
        true  
      ],  
      [  
        "",  
        {  
          "children": [  
            "sample",  
            { "children": ["__PAGE__", {}, ["$L1", "$L2", null]] },  
            [  
              "$",  
              "$L3",  
              null,  
              {  
                ...  
              }  
            ]  
          }  
        ]  
      ]  
    ]  
  ]  
]
```

Build ID

- ビルドごとの ID
- 現在のビルドと差異がある場合、MPA ナビゲーションに切り替わる

Zero Chunk を覗いてみる

```
[
  "Hd4i-EanmGD0JpIffHyHC",
  [
    [
      [
        "",
        { "children": [
          "$undefined",
          "$undefined",
          true
        ] },
      ],
      [
        "",
        {
          "children": [
            "sample",
            { "children": ["__PAGE__", {}, ["$L1", "$L2", null]] },
            [
              "$",
              "$L3",
              null,
              {
                ...
              }
            ]
          ]
        }
      ]
    ]
  ]
]
```

- 文字列で "\$" から始まるものは
特殊な Serialize/Deserialize が必要なもの

\$ の読み方

\$L3

\$ の読み方

特殊なSerializeを示すマーカ

\$L3

\$ の読み方

Serialize/Deserializeの種類

\$L3

\$ の読み方

\$L3

種類に応じた変換に用いるデータ
(他ChunkのIDなど)

特殊な Serialize / Deserialize

Serialize 可能なもの

シリアル化可能な props には以下のものがあります：

- プリミティブ
 - 文字列
 - 数値
 - bigint
 - ブーリアン
 - undefined
 - null
 - シンボル、ただし `Symbol.for` を通じてグローバルシンボルレジストリに登録されたシンボルのみ
- シリアル化可能な値を含んだ Iterable
 - 文字列
 - 配列
 - Map
 - Set
 - TypedArray と ArrayBuffer
- Date
- プレーンなオブジェクト: オブジェクト初期化子で作成され、シリアル化可能なプロパティを持つもの
- サーバアクション (server action) としての関数
- クライアントまたはサーバコンポーネントの要素 (JSX)
- プロミス

参考：

将来的にサポートされそうなもの

- FormData
- Blob
- ReadableStream
- AsyncIterator

React ドキュメント - "use client" の説明より抜粋

<https://ja.react.dev/reference/react/use-client#serializable-types>

Serialize 可能なもの

シリアル化可能な props には以下のものがあります：

- プリミティブ
 - 文字列
 - 数値
 - bigint
 - ブーリアン
 - undefined
 - null
 - シンボル、ただし `Symbol.for` を通じてグローバルシンボルレジストリに登録されたシンボルのみ
- シリアル化可能な値を含んだ Iterable
 - 文字列
 - 配列
 - Map
 - Set
 - TypedArray と ArrayBuffer
- Date
- プレーンなオブジェクト: オブジェクト初期化子で作成され、シリアル化可能なプロパティを持つもの
 - サーバアクション (server action) としての関数
 - クライアントまたはサーバコンポーネントの要素 (JSX)
 - プロミス

単純な JSON だけでは対応不可なものを含む
→ 特殊な Serialize / Deserialize が必要

考：

将来的にサポートされそうなもの

- FormData
- Blob
- ReadableStream
- AsyncIterator

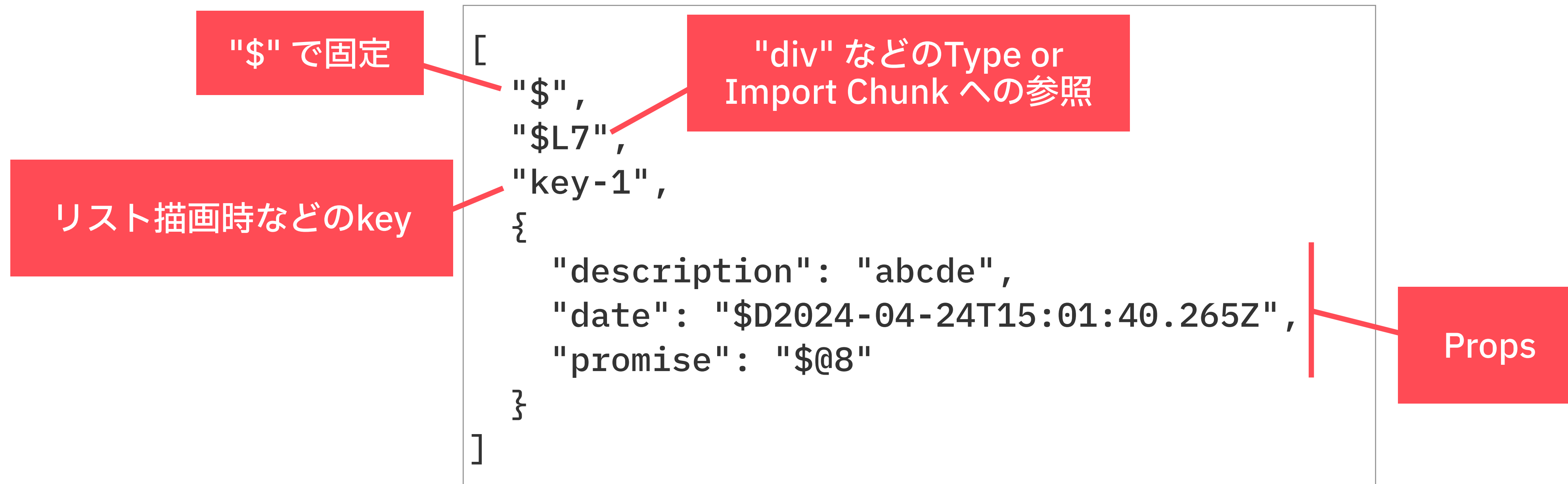
React ドキュメント - "use client" の説明より抜粋

<https://ja.react.dev/reference/react/use-client#serializable-types>

Serialize / Deserialize の具体例

React Element

- React Element は Tuple の形に Serialize される
- 先頭要素が "\$" のみの Tuple が React Element と判断される
- Tuple 内は、"\$" → Type → Key → Props の順序



一部の固定値

- JavaScript 固有の値で JSON で表現できないもの

\$Infinity	→	Infinity
\$-Infinity	→	-Infinity
\$NaN	→	NaN
\$-0	→	-0
\$undefined	→	undefined

※

ちなみに、ロジック上では\$の次の1文字で判定されており、
上記の場合 "I" "-" "N" "u" の時点で処理が分岐している。

Date

- \$D + Dateオブジェクトの toJSON() 実行結果
- Deserialize時は new Date() で Dateオブジェクトに戻される

\$D2024-04-13T08:10:15.385Z

Client Component (Lazy)

- \$L + Import Chunk ID
- Server 側では、Serialize と同時に Import Chunk も必ず作られる

Chunk

\$L7

7:I[9842,["360","static/chunks/app/sample/page-7f3480f94493b7cb.js"],"SuccessClient"]

ReactFlightServer - serializeClientReference

```
emitImportChunk(request, importId, clientReferenceMetadata);  
writtenClientReferences.set(clientReferenceKey, importId);  
if (parent[0] === REACT_ELEMENT_TYPE && parentPropertyName === '1') {  
  // If we're encoding the "type" of an element, we can refer  
  // to that by a lazy reference instead of directly since React  
  // knows how to deal with lazy values. This lets us suspend  
  // on this component rather than its parent until the code has  
  // loaded.  
  return serializeLazyID(importId);  
}
```

Import Chunk の作成/送信

\$L + ID の形にシリアライズ

Client Component (Lazy)

- Client 側では React.lazy で Import Chunk を待機する形に Deserialize

ReactFlightClient - parseModelString

```
case 'L': {  
  // Lazy node  
  const id = parseInt(value.slice(2), 16);  
  const chunk = getChunk(response, id);  
  // We create a React.lazy wrapper around any lazy values.  
  // When passed into React, we'll know how to suspend on this.  
  return createLazyChunkWrapper(chunk);  
}
```

```
function createLazyChunkWrapper<T>(  
  chunk: SomeChunk<T>,  
) : LazyComponent<T, SomeChunk<T>> {  
  const lazyType: LazyComponent<T, SomeChunk<T>> = {  
    $$typeof: REACT_LAZY_TYPE,  
    _payload: chunk,  
    _init: readChunk,  
  };  
  if (__DEV__) {  
    // Ensure we have a live array to track future debug info.  
    const chunkDebugInfo: ReactDebugInfo =  
      chunk._debugInfo || (chunk._debugInfo = []);  
    lazyType._debugInfo = chunkDebugInfo;  
  }  
  return lazyType;  
}
```

Function (Server Action)

- \$F + Server Action 情報の Chunk ID
- ビルド時点で Server Reference として関数に ID が発行されている

Chunk

```
8:{"id":"7de9e92d780a000fa18feb74a7ec9a83a73ddc8f","bound":null}
```

\$F8

```
{  
  "node": {  
    "7de9e92d780a000fa18feb74a7ec9a83a73ddc8f": {  
      "workers": {  
        "app/page": "(rsc)/./node_modules/(省略)"  
      },  
      ...  
    }  
  }  
}
```

`.next/server/server-reference-manifest.json`

Function (Server Action)

- Client 側では、該当 ID をもとに Server 側にリクエストを投げる新たな関数として Deserialize

```
case 'F': {  
  // Server Reference  
  const id = parseInt(value.slice(2), 16);  
  return getOutlinedModel(  
    response,  
    id,  
    parentObject,  
    key,  
    createServerReferenceProxy,  
  );  
}
```

"createServerReferenceProxy()" 経由で
Client 側で実行可能な関数を取得

Promise

- \$@ + Promise 情報の Chunk ID
- 参照先 Chunk には Resolve / Reject の情報が含まれる

```
const promise = new Promise((resolve) =>
  setTimeout(() => resolve("PROMISE RESOLVED"), 1000)
);
```

\$@9

Chunk

9: "PROMISE RESOLVED"

Promise

- Server 側では Serialize 対象の Promise が解決すると Stream に送信
- Client 側では ID が合致する Chunk を待機する

ReactFlightServer - serializeThenable()

```
thenable.then(  
  value => {  
    newTask.model = value;  
    pingTask(request, newTask);  
  },
```

ReactFlightClient - parseModelString()

```
case '@': {  
  // Promise  
  if (value.length === 2) {  
    // Infinite promise that never resolves.  
    return new Promise(() => {});  
  }  
  const id = parseInt(value.slice(2), 16);  
  const chunk = getChunk(response, id);  
  return chunk;  
}
```

※chunk自体がthenable

将来的に使えるかもしれない機能

Console の Server→Clientリレー

- [Flight] Instrument the Console in the RSC Environment and Replay Logs on the Client #28384
 <https://github.com/facebook/react/pull/28384>
- "enableServerComponentLogs" フラグが必要であり、Next.js v14.3.0 時点では非サポート
- 開発時の利便性向上のための機能
- Server 側での console 実行時に Stream 経由で Client 側に流す
→ Client 側で同じ console のメソッドを実行しログを出力する
- Stream Tag は "W" ("C" じゃないの…?)

Console の Server→Clientリレー

- Server 側で console のメソッドにパッチを当てている

packages/react-server/src/ReactFlightServer.js

```
0
1  if (
2    • enableServerComponentLogs &&
3    • __DEV__ &&
4    • typeof console === 'object' &&
5    • console !== null
6  ) {
7    • // Instrument console to capture logs for replaying on the client.
8    • patchConsole(console, methodName: 'assert');
9    • patchConsole(console, methodName: 'debug');
10   • patchConsole(console, methodName: 'dir');
11   • patchConsole(console, methodName: 'dirxml');
12   • patchConsole(console, methodName: 'error');
13   • patchConsole(console, methodName: 'group');
14   • patchConsole(console, methodName: 'groupCollapsed');
15   • patchConsole(console, methodName: 'groupEnd');
16   • patchConsole(console, methodName: 'info');
17   • patchConsole(console, methodName: 'log');
18   • patchConsole(console, methodName: 'table');
19   • patchConsole(console, methodName: 'trace');
20   • patchConsole(console, methodName: 'warn');
21 }
2
```

ReadableStream / AsyncIterator の Serialize

- [Flight] Encode ReadableStream and AsyncIterables #28847
<https://github.com/facebook/react/pull/28847>
- "enableFlightReadableStream" フラグが必要であり、Next.js v14.3.0 時点では非サポート
- Stream Tag に "R", "r", "X", "x", "C" が追加されている
- "R", "X" は Stream の開始 (r, x は BYOB Stream のサポート時)
"C" は Stream の終了を表す
- Server Component や Server Action で Stream を取り回せるようになるかも？

まとめ

冒頭の Stream をもう一度

- 少し読めるようになったような気がする

[illegible]

まとめ

- Stream は Tag に注意すると Chunk の種類の判別がつく
- Serialize されたデータは "\$" を見ると種類や紐づけの察しがつく
- あとは React 自体のコードを読めばなんとかなる気がする
- Serialize / Deserialize のコードを見てみると、
今後 App Router に来そうな機能が透けて見えてきて楽しい

サイボウズのフロントエンドを最高にする仲間を募集中です！

<https://cybozu.co.jp/recruit/>

チームワーク あふれる社会を創る

サイボウズは一緒に働くメンバーを待っています。

→ [募集要項・エントリー](#)